

MDMP: Managed Data Message Passing

Adrian Jackson
EPCC
The University of Edinburgh
Mayfield Road
Edinburgh
EH9 3JZ, UK
Adrian.Jackson@ed.ac.uk

Pär Strand
Plasma physics and fusion energy
Earth and Space Sciences
Chalmers University
Hörsalsvägen 9, 4th floor
Göteborg, Sweden
par.strand@chalmers.se

ABSTRACT

MDMP is a new parallel programming approach that aims to provide users with an easy way to add parallelism to programs, optimise the message passing costs of traditional scientific simulation algorithms, and enable existing MPI-based parallel programs to be optimised and extended without requiring the whole code to be re-written from scratch. MDMP utilises a directives based approach to enable users to specify what communications should take place in the code, and then implements those communications for the user in an optimal manner using both the information provided by the user and data collected from instrumenting the code and gathering information on the data to be communicated. In this paper we present the basic concepts and functionality of MDMP and discuss the performance that can be achieved using our prototype implementation of MDMP on some simple benchmark cases.

Keywords

Parallel Languages, Message Passing, MPI

1. INTRODUCTION

There are numerous new programming languages, libraries, and techniques that have been developed over the past few years to either simplify the process of developing parallel programs or provide additional functionality that traditional parallel programming techniques (such as MPI[7] or OpenMP[21]) do not provide. These include programming extensions such as Co-Array FORTRAN[20] (CAF), UPC[2], and new languages such as Chapel[3], OpenMPD[16] or XMP[8]

However, these approaches often have not had a focus of optimising the parallelisation overheads (the cost of the communications) associated with distributed memory parallelisation, and have limited appeal for the many scientific applications which are already parallelised with an existing parallelisation approach (primarily MPI) and have very large code bases which would be prohibitively expensive to re-

implement in a new parallel programming language.

The challenge of optimising parallel communications is becoming increasingly important as we approach Exascale-type high performance computers (HPC), where it is looking increasingly likely that the ratio between the computational power of a node of the computer and the relative performance of the network is going to make communications increasingly expensive when compared to the cost of calculations. Furthermore, the rise of multi-core and many-core computing on the desktop, and the related drop in single core performance, means that many more developers are going to need to exploit parallel programming to utilise the computational resources they have access to where they could have relied on increases in serial performance of the hardware they were using to maintain program performance in the past. Therefore, we have devised a new parallel programming approach, called Managed Data Message Passing (MDMP)[12], which is based on the MPI library but provides a new method for parallelising programs.

MDMP follows the directives based approach, favoured by OpenMP and other parallel programming techniques, which are translated into MDMP library function calls or code snippets, which in turn utilise communication library calls (such as MPI) to provide the actual parallel communication functionality. Using a directives based approach enables us to reduce the complexity, and therefore development code, of writing parallel programs, especially for the novice HPC programmer. However, the novel aspect of MDMP is that it allows users to specify the communication patterns required in the program but devolves the responsibility for scheduling and carrying out the communications to the MDMP functionality. MDMP instruments data accesses for data being communicated to optimise when communications happen and therefore better overlap communication and computation than is easily possible with traditional MPI programming.

Furthermore, by taking the directive approach MDMP can be incrementally added to a program that is already parallelised with MPI, replacing or extending parts of the existing MPI parallelisation without requiring any changes to the rest of the code. Users can start by replacing one part of the current communication in the code, evaluate the performance impacts, and replace further communications as required.

In this paper we outline work others have undertaken in creating new parallel programming techniques, and optimisation communications. We describe the basic issues we are looking to tackle with MDMP, and go on to describe the basic feature and functionality of MDMP, outlining the performance benefits and costs of such an approach, and highlighting the scenarios where MDMP can provide reduced communication costs for the types of communication patterns seen in some scientific simulation codes using our prototype implementation of MDMP (which implements MDMP as library calls rather than directives).

2. RELATED WORK

Recent evaluation of the common programming languages used in large scale parallel simulation code has found the majority are still implemented using MPI, with a minority also including a hybrid parallelisation through the addition of OpenMP (or in a small number of cases SHMEM) alongside the MPI functionality[25]. This highlights to us the key requirement for any new parallel programming language or technique of being easily integrated with existing MPI-based parallel codes.

There are a wide range of studies evaluating the performance of a range of different parallel languages, including Partitioned Global Address Space (PGAS) languages, on different application domains and hardware[24, 13, 23, 1]. These show that there are many approaches that can provide performance improvements for parallel programs, compared to standard parallelisation techniques on a given architecture or set of architectures. Existing parallel programming languages or models for distributed memory system provide various features to describe parallel programs and to execute them efficiently. For instance, XMP provides features similar to both CAF and HPF[19], allowing users to use either global or local view programming models, and providing easy to program functionality through the use of compiler directives for parallel functionality. Likewise, OpenMPD provided easy to program directives based parallelisation for message passing functionality, extending an OpenMP like approach to a distributed memory supercomputer.

However, both of these approaches generally require the re-writing of existing codes, or parts of existing codes, into a new languages, which we argue is prohibitively expensive for most existing computational simulation applications and therefore has limited the take-up of these different parallel programming languages or techniques by end user applications. Furthermore, both only target parts of the problem we are aiming to tackle, namely improving programmability and optimising performance. XMP and OpenMPD both aim to make parallel programming simpler, but have not direct features for optimising communications in the program (although they can enable users to implement different communication methods and therefore choose the most efficient method for themselves). PGAS languages, and other new languages, may provide lower cost communications or new models of communications to enable different algorithms to be used for a given problem, or may provide simpler programming model, but none seems to offer both as a solution for parallel programming. Also, crucially, they do not expect to work with existing MPI programs, negating the proposed benefits for the largest part of current HPC usage.

There has also been significant work undertaken looking at optimising communications in MPI programs. A number of authors have looked at compiler based optimisations to provide automatic overlapping of communications and computation in existing parallel programs[10, 6, 11]. These approaches have shown that performance improvements can be obtained, generally evaluated against kernel benchmarks such as the NAS parallel benchmarks, by transforming user specified blocking communication code to non-blocking communication functionality, and using static compiler analysis to determine where the communications can be started and finished. Furthermore, other authors have looked at communication patterns or models in MPI based parallel programs and suggested code transformations that could be undertaken to improve communication and computation overlap[4]. However, these approaches are what we would class as *coarse-grained* communication optimisation. They use only static compiler analysis to identify the communication patterns, and identify the outer bounds of where communications can occur to try and start and finish bulk non-blocking operations in the optimal places. They do not address the fundamental separation of communication and computation into different phases that such codes generally employ. Our work, outlined in this paper, is looking at *fine-grained* communication optimisations, where individual communication calls are *intermingled* with computation to truly mix communication and computation.

There has also been work on both offline and runtime identification and optimisation of MPI communications, primarily for collective communication[9, 14, 5], or other auto-tuning techniques such as optimising MPI library variables[22] or individual library routines[17]. All these approaches have informed the way we have constructed MDMP.

We believe that the work we have undertaken is unique as it brings together attempts to provide simple message passing programming which fine-grained communication optimisation along with the potential for runtime auto-tuning of communication patterns into a single parallel programming tool.

3. COMMUNICATION OPTIMISATION

Whilst there is a very wide range of communication and computational patterns in parallel programs, a large proportion of common parallel applications use regular domain decomposition techniques coupled with *halo* communications to exploit parallel resources. As shown in Figure 3, which is a representation of a Jacobi-style stencil based simulation method, many simulations undertake a set of calculations that iterate over a n-dimensional array, with a set of communications to neighbouring processes every iteration of the simulation.

The core computational kernel of a simple Jacobi style simulation, as illustrated in the previous paragraph, can be implemented as shown in Figure 3 (undertaking a 2d simulation).

It is evident from the above code that, whilst it has been optimised to use non-blocking communications, the communication and computation parts of the simulation are performed separately, with no opportunity to overlap commu-

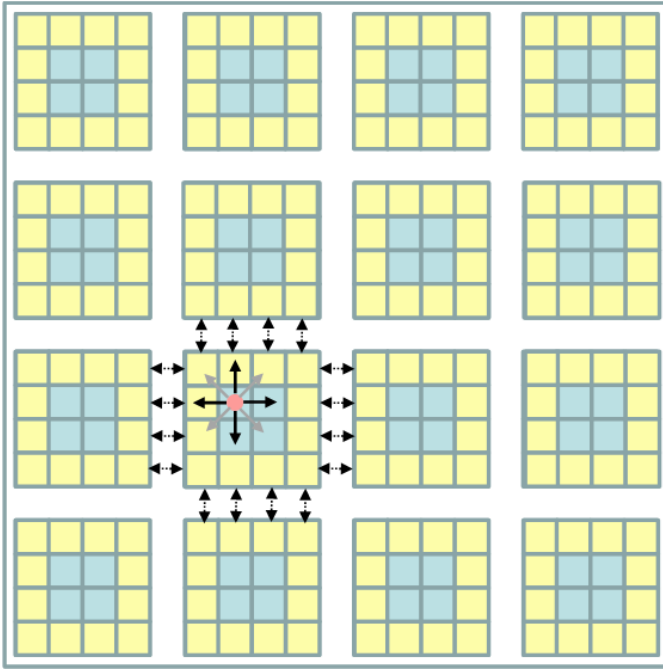


Figure 1: Representation of a common communication pattern for parallel simulations

nications and computations. In practice this means that the application will only be using the communication network to send and receive data in short bursts, leaving it idle whilst computation is being performed.

Many large scale HPC resources are used by a large number of running applications at any one time, which may help to ensure that the overall usage of the interconnect is high, even though individual applications often utilise it in a *bursty* manner. However, that still will not be true of the part of the network dedicated to the individual application, only to the load of the network overall. Furthermore, when considering the very largest HPC resources in the world, and including the proposed Exascale resources, there are often only a handful of applications utilising the resource at any one time. Therefore, enabling applications to effectively utilise the network, especially the *spare* resources that the current separated communication and computation patterns engender, is likely to be beneficial to overall application performance and resource utilisation (provided that the cost of doing this is not significant).

It is possible to split the sends and receives in the previous example and place them around the computation rather than just before the computation, using the non-blocking functionality, to further ensure that more optimal communications are occurring. However, this still does not allow for overlapping communication and computation because the computation is still occurring in a single block, with communications outside this block.

For developers to ensure that communications and computations are truly mixed would require further code modifi-

```
for (iter=1; iter<=maxiter; iter++){
    MPI_Irecv(&old[0][1], NP, MPI_FLOAT, prev, 1,
              MPI_COMM_WORLD, &requests[0]);
    MPI_Irecv(&old[MP+1][1], NP, MPI_FLOAT, next, 2,
              MPI_COMM_WORLD, &requests[1]);
    MPI_Isend(&old[MP][1], NP, MPI_FLOAT, next, 1,
              MPI_COMM_WORLD, &requests[2]);
    MPI_Isend(&old[1][1], NP, MPI_FLOAT, prev, 2,
              MPI_COMM_WORLD, &requests[3]);
    MPI_Waitall(4, requests, statuses);
    for (i=1; i<MP+1; i++){
        for (j=1; j<NP+1; j++){
            new[i][j]=0.25*(old[i-1][j]+old[i+1][j]+
                           old[i][j-1]+old[i][j+1] - edge[i][j]);
        }
    }
    for (i=1; i<MP+1; i++){
        for (j=1; j<NP+1; j++){
            old[i][j]=new[i][j];
        }
    }
}
```

Figure 2: MPI implementation of a simple 2d Jacobi style computation, implemented in C using MPI

cations, as shown in the code example in Figure 3 (which implements a strategy of sending data as soon as it has been computed).

Whilst the code implemented in Figure 3 will enable the mixing of communication and computation, ensuring that data is sent as soon as it is ready to be communicated and potentially ensuring better utilisation of the communication network, it has come at the cost of considerable code *mutation*, requiring developers to undertake significant code optimisations. As well as the damage to the readability and maintainability of the code that this causes, it also means that a code has been significantly changed for a potentially architecturally dependent optimisation, i.e. an optimisation that may be beneficial on one or more current HPC systems but may not be beneficial on other or future HPC systems.

We are proposing MDMP as a mechanism for implementing such optimisations without the requirement to significantly change users codes, or the need to tailor codes to a specific platform, as the MDMP functionality can implement communications in the most optimal form for the application and hardware currently being used.

4. MDMP

To re-iterate the challenges for MDMP that we have previously discussed, MDMP is designed to address the following issues:

- Work with existing MPI based codes
- Provide framework for optimisation communications
- Simplify parallel development

```

for (iter=1;iter<=maxiter; iter++){
    requestnum = 0;
    for (j=0;j<NP;j++){
        MPI_Irecv(&tempprev[j], 1, MPI_FLOAT, prev, 1,
            MPI_COMM_WORLD, &requests[requestnum]);
        requestnum++;
        MPI_Irecv(&tempnext[j], 1, MPI_FLOAT, next, 2,
            MPI_COMM_WORLD, &requests[requestnum]);
        requestnum++;
    }
    for (i=1;i<MP+1;i++){
        for (j=1;j<NP+1;j++){
            new[i][j]=0.25*(old[i-1][j]+old[i+1][j]+
                old[i][j-1]+old[i][j+1] - edge[i][j]);
            if(i == MP){
                MPI_Isend(&new[i][j], 1, MPI_FLOAT, next, 1,
                    MPI_COMM_WORLD, &requests[requestnum]);
                requestnum++;
            } else if(i == 1){
                MPI_Isend(&new[i][j], 1, MPI_FLOAT, prev, 2,
                    MPI_COMM_WORLD, &requests[requestnum]);
                requestnum++;
            }
        }
    }
    for (i=1;i<MP+1;i++){
        for (j=1;j<NP+1;j++){
            old[i][j]=new[i][j];
        }
    }
    MPI_Waitall(requestnum, requests, statuses);
    if(prev != MPI_PROC_NULL){
        for (j=1;j<NP+1;j++){
            old[0][j] = tempprev[j-1];
            old[MP+1][j] = tempnext[j-1];
        }
    }
    if(next != MPI_PROC_NULL){
        for (j=1;j<NP+1;j++){
            old[MP+1][j] = tempnext[j-1];
        }
    }
}

```

Figure 3: Intermingled MPI implementation of a simple 2d Jacobi style computation

MDMP uses a directives based approach, relying on the compiler to implement the actual message passing functionality based on the users' instructions. Compiler directives are used, primarily, to address the third point above, namely ease of use. We provide functionality that can be easily enabled and disabled in an application, hides some of the complexities of current MPI programming (such as providing message tags, error variables, communicators, etc...) that often complicate development for new users of MPI, and also provides some flexibility to the user over the type and level of message optimisation used.

The MDMP directives are translated into code snippets and library calls by the MDMP-enabled compiler, either directly in the equivalent non-blocking MPI calls (which simply mimics the communication that would have been implemented directly by the user) or to further optimised MPI communications or other another communication library as appropriate on the particular hardware being used. This enables MDMP to target different communication libraries transparently to the developer for a given HPC system. Also, crucially the ability to target MPI communications means that MDMP functionality can be added to existing MPI-parallelised programs, either as additional functionality or to replace existing MPI functionality, without requiring the program to be completely changed into a new programming language or utilise a new message-passing (or other) communication library.

However, simply using directives for programming message passing will not optimise the communication that are undertaken by a program. Therefore, MDMP provides not only directives to specify the communications to be undertaken in the program but also directives to specify *communication regions*. Communication regions define the areas of code where the data that is to be sent and received is worked on, and where communications occur, so that MDMP can, at runtime, examine the data access patterns and undertake communications at the optimal time to intermingle communications and computations and therefore better utilise the communication network.

The optimisation of communications is based on runtime functionality that monitors the reads and writes of data that has been specified as communication data (data that will be sent or received). As any data monitoring entails some runtime overheads the communication region specifies the scope of the data monitoring to ensure it is only performed where required (i.e. where communications are occurring). Any data that is specified by the users and being involved in send or receives is tracked so each read and writing in a communication region is recorded and the number of reads and writes that have occurred when the send or receive happens is evaluated. This data, the number of reads and writes that have occurred for a particular piece of data when it comes to be sent or written over by a receive, can then be used in any subsequent iterations of the computation to launch the communication of that data once it is ready to be communicated.

Communications are triggered for any given piece of data as follows:

```

#pragma commregion
for (iter=1;iter<=maxiter; iter++){
#pragma recv(old[0][0], NP, prev)
#pragma recv(old[MP+1][1], NP, next)
#pragma send(old[MP][1], NP, next)
#pragma send(old[1][1], NP, prev)
    for (i=1;i<MP+1;i++){
        for (j=1;j<NP+1;j++){
            new[i][j]=0.25*(old[i-1][j]+old[i+1][j]+
                old[i][j-1]+old[i][j+1] - edge[i][j]);
        }
    }
    for (i=1;i<MP+1;i++){
        for (j=1;j<NP+1;j++){
            old[i][j]=new[i][j];
        }
    }
}
#pragma commregionfinished

```

Figure 4: MDMP implementation of a simple 2d Jacobi style computation with optimised communication

- last write occurs (sends)
- last read and/or write occurs (receives)

Using this functionality we can implement a communication pattern that intermingles communication and computation for the example code shown in Figure 3, as shown in Figure 4.

When compiled with an MDMP-enabled compiler, the code in Figure 4 will be processed by the compiler and non-blocking sends and receives inserted where the `send` and `recv` directives are placed. The compiler then looks through the code associated with the communicating region (between `commregion` and `commregionfinished`) and replaces any variable reads or writes linked to those sends and receives by MDMP code which will perform the reads and writes and also record those reads and writes occurring.

Compiler based code analysis for data accesses will be straightforward for many applications, however we recognise that there will be a number of scenarios, such as when pointers are heavily used in C or FORTRAN, or possibly where pre-processing or function pointers or conditional function calls are used, where it will not be possible for the compiler to access where the data accesses for a particular `send` or `recv` occur. In that situation MDMP will revert to simply inserting the basic MPI function calls required to undertake the specified communication and not perform the optimised message passing functionality. Whilst this negates the possibility of optimising the communications, it will not add any overheads to the program compared to the standard MPI performance a developer would experience, and it does still leave scope for the MDMP functionality to target communication libraries other than MPI to enable optimisation for users would require them to modify their code, if such functionality is available.

Furthermore, whilst we are not investigating such functionality at the moment, the design of MDMP means that it can also undertake auto-tuning or other runtime activities to optimise communication performance for users beyond the intermingling communication optimisations we have already discussed. For instance, MDMP could implement additional helper threads that enable progression of communications whilst the main program is undertaking calculations, albeit at the cost of utilising a computational core for that purpose. It could also evaluate different communication optimisations at runtime to auto-tune the performance of the program whilst it is running.

A difference between the MPI functionality that a developer would add to a code like the one we have been considering and the functionality that MDMP implements is that where intermingling of communications is undertaken MDMP will be sending lots of single element (or small numbers of elements) messages between processes rather than a single message with all the data in it. In general, MPI performs best when small numbers of large messages are used, rather than large numbers of small messages. This is because in the case that large numbers of small message are sent the communication costs are dominated by the latency costs of each message, whereas using small numbers of large messages reduces the overall number of message latencies that are incurred. We recognise the fact the the MDMP functionality may not be optimal in terms of the overall message costs associated with communications but we are assuming that this penalty will be negated by the benefits associated with more consistent use of the network and less concentrated nature of the communication and computation patterns. However, this is something that is investigated in our performance analysis of MDMP, and as with other previously discussed potential problems with MDMP if it is impacting performance the optimised message passing can be disabled at compile time.

We also recognise that MDMP functionality does not come without a cost to the performance of the program. MDMP is adding additional computational requirements above those specified in the user program, and also requires additional memory to store data associated with the communications (such as the counters that record the reads and writes to variables). The premise behind the optimised message passing functionality we are aiming for is that communications are much more expensive than computations for an application on a modern HPC machine, and this relationship is likely to get worse for future HPC machines. If this is the case then adding additional computational requirements can be acceptable provided the communication costs are reduced through this addition of extra computation. We have evaluated the performance impact of MDMP, and the communication verses computation trade-off on current HPC architectures where MDMP becomes beneficial, through benchmarking of our software, described in the next section. However, we are still working on minimising the memory requirements for MDMP, as this will be important to ensure MDMP is usable on current and future HPC systems. Furthermore, we should remember that MDMP can be used as a simpler programming alternative to MPI with the optimised message passing functionality turned off at compile time, thereby removing all of these overheads if they are not beneficial for a given application of HPC platform.

5. EXPERIMENTAL SETUP

We evaluated the performance of the MDMP compared to standard C and MPI codes. We undertook our evaluation using a range of common large scale HPC platforms and a set of simple, *kernel* style, benchmarks. We have only evaluated the functionality using 2 nodes on each system, primarily testing the communications between a pair of communicating processors, one on each node.

5.1 Computing Resources

We used three different large scale HPC machines to benchmark performance. The first was a **Cray XE 6**, HECToR, is the UK National Supercomputing Service consists of 2816 nodes, each containing two 16-core 2.3 GHz *Interlagos* AMD Opteron processors per node, giving a total of 32 cores per node, with 1 GB of memory per core. This configuration provides a machine with 90,112 cores in total, 90TB of main memory, and a peak performance of over 800 TFlop/s. We used the PGI FORTRAN compile on HECToR, compiled with the `-fastsse` optimisation flag.

The second was a **Bullx B510**, HELIOS, which is based on Intel Xeon processors. A node contains 2 Intel Xeon E5-2680 2.7 GHz processors giving 16-cores and 64 GB memory. HELIOS is composed of 4410 nodes, providing a total of 70,560 cores and a peak performance of over 1.2 PFlop/s. The network is built using Infiniband QDR non-blocking technology and is arranged using a fat-tree topology. We used the Intel FORTRAN compiler on HELIOS, compiling with the `-O2` optimisation flag.

The final resource was a **BlueGene/Q**, JUQUEEN at Forschungszentrum Juelich. JUQUEEN is a IBM BlueGene/Q system based on the IBM POWER architecture. There are 28 racks composed of 28,672 nodes giving a total of 458,752 compute cores and a peak performance of 5.9 PFlop/s. Each node has an IBM PowerPC A2 processor running at 1.6 GHz and containing 16 SMT cores, each capable of running 4 threads, and 16 GB of SDRAM-DDR3 memory. IBM's FORTRAN compile, xlf90, was used on JUQUEEN, compiling using the `-O2` optimisation flag.

6. PERFORMANCE RESULTS

We have been evaluating MDMP functionality using a number of different benchmarks. Initially we tested the performance impact of instrumenting data reads and writes on a non-communicating code, the STREAMS [18] benchmark, with the results presented in the first subsection below. After this we evaluated the communications performance of MDMP verses communication implemented directly with MPI, the results of these evaluations are in the second subsection below.

Each of the STREAMS benchmark tests were repeated 10 times and an average runtime calculated. For the communications benchmarks each operation was run 100 times, and each benchmark as repeated 3 times with the average time taken.

6.1 Reference Implementation

Whilst we have, in previous sections in this paper, outlined the principles of MDMP and how it designed to work, we

Table 1: C STREAM benchmark within a MDMP inside a communicating region (times in seconds)

Operation	Original	MDMP	Optimised MDMP
Int Assign	0.000008	0.000172	0.000076
Db Assign	0.000010	0.000163	0.000084
Db Copy	0.000009	0.000295	0.000151
Db Scale	0.000016	0.000296	0.000152
Db Add	0.000031	0.000447	0.000228
Db Triad	0.000042	0.000439	0.00238

do not yet have a full compiler based implementation of this functionality. We have designed and implemented the runtime functionality that any compiler would added to a code when encountering MDMP pragmas, but have not yet implemented the compiler functionality. Therefore, for this performance evaluation we are using benchmarks where the MDMP functionality has been implemented directly in the benchmark.

We have implemented two versions of MDMP; the first version implements all the required functionality within function calls to the MDMP library. This includes data stores and lookups for all the data marked as being communicated within an MDMP communicating region.

The second version implements exactly the same functionality but uses pre-processor macros to insert the required code directly into the source code, thereby removing the need for function calls at every point MDMP is used. In the benchmark results this is named *Optimised MDMP*

Work is currently ongoing to implement a compiler based solution, utilising the LLVM[15] compiler infrastructure, to enable us to target all the main HPC computer languages with a single, full reference implementation.

6.2 STREAM Benchmark Results

STREAMS is often used to evaluate the memory bandwidth of computer hardware, and therefore was chosen as it will highlight any impact on the memory access and update efficiencies of computations when MDMP is added to a code.

The performance of the STREAMS benchmark was evaluated on all three of the hardware platforms we had available to us, although we are only presenting the results in the following tables from the Cray XE6 because, although the performance of the benchmark varies between machines, the relative performance difference between the original implementation of STREAMS and our MDMP implementations does not change significantly between these platforms.

We are reporting the results from a single process running on an otherwise empty node on the Cray XE6. Table 6.2 (where Int stands for Integer and Db stands for Double) outlines the performance of the two versions of MDMP verses the original STREAMS code when the fully communicating functionality of MDMP is enabled and we have forced the MDMP library to treat each variable in the arrays being processed as if they were being communicated (i.e. we are fully tracking all the reads and writes to these array entries even though no communications are occurring).

Table 2: C STREAM benchmark within a MDMP outside communicating region (times in seconds)

Operation	Original	MDMP	Optimised MDMP
Int Assign	0.000003	0.000103	0.000017
Db Assign	0.000007	0.000079	0.000021
Db Copy	0.000009	0.000118	0.000025
Db Scale	0.000016	0.000126	0.000020
Db Add	0.000030	0.000188	0.000025
Db Triad	0.000038	0.000194	0.000025

We can see from Table 6.2 that the MDMP functionality does have a significant impact on the overall runtime of all the benchmarks, adding around an order of magnitude increase to the runtime of the benchmark. We would expect any benchmark like this, where no communications are involved and therefore there is no optimisation for MDMP to perform, to be detrimentally impacted by the additional functionality added in the MDMP implementation. However, we can see that the optimised implementation of MDMP does not have as significant an impact as the original MDMP implementation. As this is the first optimisation we have done to the MDMP functionality we are hopeful there is further scope for optimising the performance of MDMP and reducing the computational impact of the MDMP functionality.

Furthermore, it is worth re-iterating that in this benchmark we are forcing MDMP to mark and track all the data in the STREAMS benchmark as if it is to be communicated. MDMP is not designed to be beneficial for scenarios such as this, it is primarily designed to be useful in scenarios where a small amount of data (compared to the overall amount computed upon) is sent each iteration. In a scenario such as the one this benchmark mimics (where all the data used in the computation is also communicated) MDMP should simply compiled down to the basic MPI calls as they would be more efficient in this scenario. MDMP is designed to enable users to try different communication strategies, such as simply using the plain MPI calls, or trying to vary the amount communication and computation intermingling, which enables users to experiment and evaluate which will give them the best performance for their application and use case. Indeed, such functionality could also be built into the runtime of MDMP, enabling auto-tuning of the choice of communication optimisation on the fly.

We also ran the same benchmark with the code marked as outside a communicating region. In this scenario, whilst the MDMP functionality has been enabled for all the variables in the calculation, the absence of a communicating region disables, at runtime, any data tracking associated with the variables. Table 6.2 presents the results for this benchmark. We can see that the cost of the MDMP functionality has been substantially reduced, and indeed if we used the optimised functionality where the MDMP function calls have been removed and replaced with pre-processed code the MDMP performance is extremely close to the plain benchmark codes' performance. This confirms that the MDMP functionality can be constructed in such a way as not to have a significant adverse impact on the key computational kernels of a code outside the places that communications are occurring.

However, we can see from the results that if communications are present there is a significant performance impact on the data that is tracked by the MDMP functionality. Our assumption is that the computational cost associated by MDMP can be more than offset by the reduction in communication costs for a program, but clearly this is dependent on the ratio between communications and computations for a given kernel, and the ratio of relative costs (in terms of overall runtime) of a communication versus a computation.

We evaluate the performance impact versus the communication cost savings in the next subsection, where we analyse some communication benchmarks.

6.3 Message Passing Results

We have constructed four simple benchmarks to evaluate MDMP against MPI. The first is a **PingPong** benchmark where a process sends a message to another process who copies the received data from the receive buffer into its send buffer and sends it back to the first process, who performs the same copying process and sends it back again. This pattern is repeated many times and the time for the communications are recorded. The benchmark can send a range of message sizes. For the reference MPI benchmark only a single message is sent each iteration of the benchmark containing the fully amount of data to be sent. For the MDMP version the `send` and `recv` functionality specifies the single message to be sent and received, and performs the send and receive on the first iteration of the benchmark but on subsequent iterations of the benchmark the MDMP functionality identifies when each element of the message data is ready to be sent (through tracking the data copying process between the send and receive buffers) and sends individual elements when they are ready to go. This will mean that for a run of the benchmark using a message of 1000 elements in size the MPI version will send one message between processes whereas the MDMP version will send 1000 messages (apart from on the first iteration where it will only send one message).

The second benchmark, called **SelectivePingPong** alters the basic PingPong benchmark we have already described by performing the same functionality but only sending a portion of the overall data owned by a process in the messages. It is possible to vary both the overall size of data each process has, and the amount of that data that is sent, for instance you could have each process having an array that is 100 elements long but only the first 10 and last 10 elements are sent in the PingPong messages. This benchmark is designed to investigate the performance impact of varying the overall data in a computation and the amount that is being communicated via MDMP.

The third benchmark, called **DelayPingPong**, also alters the basic PingPong benchmark by adding a delay in the loop that copies the data from the receive buffer to the send buffer. This delay is variable and is designed to simulate some level of computational work being undertaken during what would be the main computational loop for a computational kernel using MDMP. The delay is performed by a routine which iterates through a loop adding an integer to a double a specified number of times (delay elements).

The final benchmark, **SelectiveDelayPingPong**, combines the second and third benchmarks meaning the PingPong process can contain both user defined delay in the data copy loop and a selective amount of data to be transferred.

Figure 5a demonstrates the cost of MDMP compared to plain MPI where there is no scope for communication and computational overlaps. The runtime for MDMP increases more or less linearly as the size of the data to be transferred increases, whereas the runtime for MPI stays relatively constant.

However, if we examine Figure 5b we can see that MDMP begins to see some benefits over MPI when the delay added to the data copy routine is increased. The JUQUEEN and HECToR MDMP is faster than MPI when the delay elements are around 1000 and 800 elements respectively, although for HELIOS MPI is always faster than MDMP (albeit with a smaller gap in performance between the two methods).

If not all the data that is copied between buffers is sent, as in the case of the SelectivePingPong benchmark shown in Figure 6a, then in comparison to the normal PingPong benchmark the overall difference in performance is reduced between MPI and MDMP although MDMP is still more costly than MPI.

Finally, the combined benchmark, results shown in 6a where 1024 overall data elements are processed and either 1 or 32 elements are sent with variable amounts of delays, highlight where MDMP can improve performance. When only one element is being sent then all it requires is 16 floating point adds between communications (16 delay elements)¹ to enable MDMP to optimise communications. If 32 elements are being sent then around 32 floating point adds are required to enabling the communication hiding that MDMP enables to provide a performance benefit.

Whilst these benchmarks are beneficial in enabling us to evaluate MDMP performance we recognise that a more realistic benchmark that evaluates MDMP performance against real kernel computations would also be useful as it would enable us to evaluate the overall impact of MDMP on cache, memory, and processor usage for real applications. We are in the process of undertaking such benchmarks at the moment but unfortunately do not have these results in time for this paper submission.

7. CONCLUSIONS

We have outlined a novel approach of message passing programming on distributed memory HPC architectures and demonstrated that, given a reasonable level of computations to the communications to be performed, MDMP can reduce the overall cost of communications and improve application performance. We are aware the MDMP presents performance risks for parallel programs, including impacting cache and memory usage, and consuming additional memory. However, we believe the ability to enable and disable MDMP optimisations, and the potential benefits to ease of

use and programmability from MDMP, make this approach a sensible one to investigate future message passing programming.

We are currently working on a full compiler implementation of MDMP, including a formal MDMP language definition, and more involved benchmarks to evaluate MDMP in much more detail.

8. ACKNOWLEDGMENTS

Part of this work was supported by an e-Science grant from Chalmers University.

Part of this work was supported by the Nu-FuSE project which is funded by EPSRC through G8 Multilateral Research Funding Nu-FuSE grant EP/J004839/1.

9. REFERENCES

- [1] F. Blagojević, P. Hargrove, C. Iancu, and K. Yelick. Hybrid pgas runtime support for multicore nodes. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM.
- [2] W. W. Carlson, J. M. Draper, and D. E. Culler. S-246, 187 introduction to upc and language specification.
- [3] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, Aug. 2007.
- [4] A. Danalis, K. yong Kim, L. Pollock, and M. Swamy. Transformations to parallel codes for communication-computation overlap. In *In Supercomputing 2005*, page 58, 2005.
- [5] A. Faraj, X. Yuan, and D. Lowenthal. Star-mpi: self tuned adaptive routines for mpi collective operations. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 199–208, New York, NY, USA, 2006. ACM.
- [6] L. Fishgold, A. Danalis, L. Pollock, and M. Swamy. An automated approach to improve communication-computation overlap in clusters. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 290–290, Washington, DC, USA, 2006. IEEE Computer Society.
- [7] M. Forum. MPI: A message-passing interface standard. available at: <http://www.mpi-forum.org>.
- [8] X. S. W. Group. Specification of xcalablemp, version 1.1, november 2012. available from: <http://www.xcalablemp.org/spec/xmp-spec-1.1.pdf>.
- [9] T. Hoefer and T. Schneider. Runtime detection and optimization of collective communication patterns. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 263–272, New York, NY, USA, 2012. ACM.
- [10] C. Hu, Y. Shao, J. Wang, and J. Li. Automatic transformation for overlapping communication and computation. In J. Cao, M. Li, M.-Y. Wu, and J. Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 210–220. Springer Berlin Heidelberg, 2008.
- [11] C. Iancu, W. Chen, and K. Yelick. Performance

¹Actually there are $1023 * 16$ delay elements as there is a delay per array element

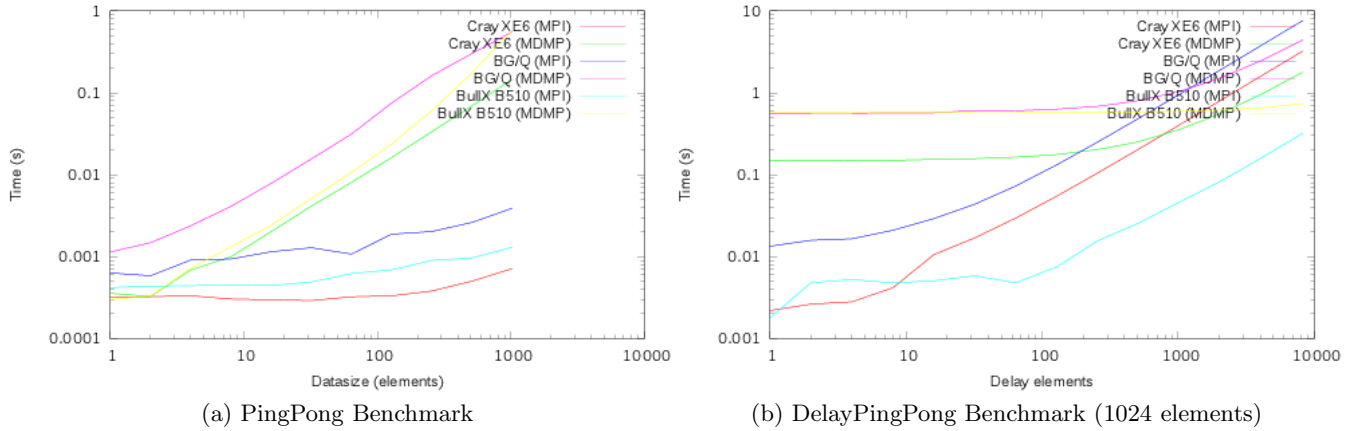


Figure 5: Runtime of PingPong and DelayPingPong benchmarks using up to 1024 data elements

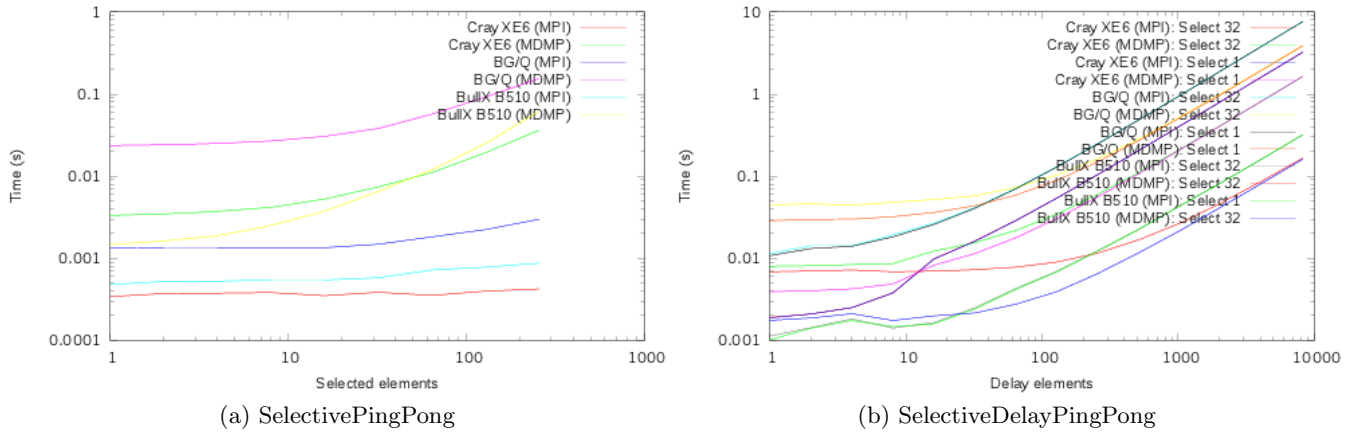


Figure 6: Runtime of benchmarks with 1024 data elements, varying the number of selected elements or the number of delay elements

portable optimizations for loops containing communication operations. In *Proceedings of the 22nd annual international conference on Supercomputing, ICS '08*, pages 266–276, New York, NY, USA, 2008. ACM.

- [12] A. Jackson and P. Strand. MDMP: Managed Data Message Passing. In *Exascale Applications and Software Conference 2013*, April 2013.
- [13] H. Jin, R. Hood, and P. Mehrotra. A practical study of upc using the nas parallel benchmarks. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, PGAS '09*, pages 8:1–8:7, New York, NY, USA, 2009. ACM.
- [14] A. Knapfer, D. Kranzlmaller, and W. Nagel. Detection of collective mpi operation patterns. In D. Kranzlmaller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 259–267. Springer Berlin Heidelberg, 2004.
- [15] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis &

transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, CGO '04*, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

- [16] J. Lee, M. Sato, and T. Boku. Openmpd: A directive-based data parallel language extension for distributed memory systems. In *Parallel Processing - Workshops, 2008. ICPP-W '08. International Conference on*, pages 121–128, 2008.
- [17] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning gemm for gpus. In *Proceedings of the 9th International Conference on Computational Science: Part I, ICCS '09*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.
- [18] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.
- [19] A. Müller and R. Rühl. Extending high performance fortran for the support of unstructured computations.

- In *Proceedings of the 9th international conference on Supercomputing*, ICS '95, pages 127–136, New York, NY, USA, 1995. ACM.
- [20] R. W. Numrich and J. Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24(2):4–17, Aug. 2005.
 - [21] OpenMP. Openmp architecture review board. openmp fortran application program interface, version 1.1. available from: <http://www.openmp.org>.
 - [22] S. Pellegrini, T. Fahringer, H. Jordan, and H. Moritsch. Automatic tuning of mpi runtime parameter settings by using machine learning. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 115–116, New York, NY, USA, 2010. ACM.
 - [23] H. Shan, F. Blagojević, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A programming model performance study using the nas parallel benchmarks. *Sci. Program.*, 18(3-4):153–167, Aug. 2010.
 - [24] T. Wen, J. Su, P. Colella, K. Yelick, and N. Keen. An adaptive mesh refinement benchmark for modern parallel programming languages. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 40:1–40:12, New York, NY, USA, 2007. ACM.
 - [25] M. B. Xu Guo. Prace deliverble d7.4.3: Tier-0 applications and systems usage, may 2012.